On the power of deep pushdown stacks

A. Arratia Quesada*
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya, Barcelona, Spain

I.A. Stewart
Department of Computer Science, Durham University,
Science Labs, South Road, Durham DH1 3LE, U.K.

Abstract

Inspired by recent work of Meduna on deep pushdown automata, we consider the computational power of a class of basic program schemes, NPSDS$_s$, based around assignments, while-loops and non-deterministic guessing but with access to a deep pushdown stack which, apart from having the usual push and pop instructions, also has deep-push instructions which allow elements to be pushed to stack locations deep within the stack. We syntactically define sub-classes of NPSDS$_s$ by restricting the occurrences of pops, pushes and deep-pushes and capture the complexity classes **NP** and **PSPACE**. Furthermore, we show that all problems accepted by program schemes of NPSDS$_s$ are in **EXPTIME**.

1 Introduction

In automata theory, there is a variety of machine models, both restricting and enhancing pushdown automata, such as finite-turn pushdown automata [7, 23] and two-pushdown automata [13, 14] (the former accept a proper sub-class of the context-free languages and the latter are as powerful as Turing machines). However, rarely is there a model based on modifications of pushdown automata capturing a class of languages lying between the classes of the context-free and the context-sensitive languages. In an attempt to remedy this situation, Meduna [15] recently introduced deep pushdown automata and showed that these models coincide with Kasai's state grammars [11], and thus give rise to an infinite hierarchy of languages lying between the classes of the context-free and the context-sensitive languages. Meduna's deep pushdown automata pop and push items from and to their stack in the standard way; however, they also have the facility to insert (but not read) strings at some position within the stack.

Inspired by Meduna's machine model, we consider in this paper the formulation of program schemes with access to a deep pushdown stack. Program schemes work on arbitrary finite structures (rather than just strings) and are more computational in flavour than are formulae of logics studied in finite model theory and descriptive

*Research partially supported by Spanish Government MICINN under Projects: SESAAME (TIN2008-06582-C03-02) and SINGACOM (MTM2007-64007)

complexity, yet they remain amenable to logical manipulation. The concept of a program scheme originates from the 1970's with work of, for example, Constable and Gries, Friedman, and Hewitt and Paterson [2, 6, 16], and complexity-theoretic considerations of such program schemes were subsequently studied by, for example, Harel and Peleg, Jones and Muchnik, and Tiuryn and Urzyczyn [8, 10, 22].

Of relevance to our work here is [1] where it was shown that a basic class of program schemes, in which there are assignments, while-loops and non-deterministic guessing, augmented with access to a stack, where these program schemes work on ordered data, is such that the class of problems accepted is **P**. This characterization should be compared with Cook's result from [3] that a non-deterministic pushdown automaton can be simulated by a deterministic pushdown automaton, with the class of languages accepted by such automata being **P**. Moreover, as with Cook's scenario, the above program schemes can be simulated by (deterministic) program schemes in which guessing is not allowed. (There has also been a consideration of these program schemes on unordered data in [1, 20].) An interesting aspect of program schemes with a stack is that there is potentially an unlimited amount of memory available to a computation (in the form of stack storage) yet (as the results of [1] show) the problems accepted by such program schemes can all be solved in polynomial-time (on a Turing machine; in fact, it appears that some problems in **P** require exponential time when 'time' is defined to be the number of instructions executed by a program scheme with a stack that solves the problem [20]).

As mentioned above, in this paper it is our intention to consider the computational power of basic program schemes when augmented with a deep pushdown stack. A deep pushdown stack is such that the usual pops and pushes are available as well as an additional instruction which allows elements to be written to locations deep in the stack but only so that the depth at which such a location lies (from the top of the stack) is bounded by some polynomial in the size of the input structure; that is, deep-pushes cannot be to locations too deep within the stack. Note that we only have deep-pushes, not deep-pops; for if we had both then we would have, essentially, access to a stack the top (polynomial) portion of which could be used as an array (program schemes with arrays have been considered in [19, 21]). Our goal is to classify the computational power of basic program schemes with deep pushdown stacks in comparison with the standard complexity classes of computational complexity. Such complexity classes are all defined with regard to ordered data (more specifically, strings of symbols) and consequently we always assume that our program schemes work on finite structures augmented with an ordering of the data (this statement will be made explicit in the subsequent definitions).

Of course, the results of [1] show that any problem in **P** can be accepted by some program scheme from our class of program schemes with access to a deep pushdown stack, which we call NPSDS_s. It turns out that we can (syntactically) define subclasses of the class of program schemes NPSDS_s, obtained by restricting how the occurrences of pops, pushes and deep-pushes are structured, capturing the complexity classes **NP** and **PSPACE**. Furthermore, we show that all problems accepted by program schemes of NPSDS_s are in **EXPTIME**.

Let us end this introduction by remarking that there do exist programming languages with facilities for the manipulation of elements deep within a stack. One such is the programming language Push [17], specifically defined for use in genetic and evo-

lutionary computational systems, where the instructions YANK and SHOVE allow deep access to stack elements, by means of integer indices.

Our basic definitions (relating to logic and program schemes) are given in Section 2 before we consider some simple restrictions of our program schemes in Section 3. We capture the complexity class **NP** with a sub-class of our program schemes in Section 4, and the complexity class **PSPACE** in Section 5. We show that any problem accepted by a program scheme of NPSDS$_s$ is in **EXPTIME** in Section 6, before presenting our conclusions and directions for further research in Section 7.

2 Basic definitions

Throughout, a *signature* τ is a tuple $\langle R_1, \ldots, R_r, C_1, \ldots, C_c \rangle$, where each R_i is a relation symbol, of arity a_i, and each C_j is a constant symbol. A *finite structure* $\mathcal{A}$ *over the signature* τ, or τ*-structure*, consists of a *universe* or *domain* $|\mathcal{A}| = \{0, 1, \ldots, n-1\}$, for some natural number $n \geq 2$, together with a relation R_i of arity a_i over $|\mathcal{A}|$, for every relation symbol R_i of τ, and a constant $C_j \in |\mathcal{A}|$, for every constant symbol C_j of τ (by an abuse of notation, we do not distinguish between constants or relations and constant or relation symbols). A finite structure $\mathcal{A}$ whose domain is $\{0, 1, \ldots, n-1\}$ has *size* n, and we denote the size of $\mathcal{A}$ by $|\mathcal{A}|$ also (this does not cause confusion). The set of all finite structures over the signature τ is denoted STRUCT(τ). A *problem* over some signature τ consists of a subset of STRUCT(τ) which is closed under isomorphisms. Throughout, all our structures are finite. We insist that the binary relation symbol *succ* and the two constant symbols 0 and *max* never appear in any signature. The binary relation *succ* is only ever interpreted as a *successor relation*, that is, as a relation of the form:

$$\{(u_0, u_1), (u_1, u_2), \ldots, (u_{n-2}, u_{n-1}) : u_i \neq u_j, \text{ for } i \neq j\},$$

on some domain of size n, and 0 (resp. *max*) is only ever interpreted as the minimal (resp. maximal) element u_0 (resp. u_{n-1}) of the successor relation *succ*.

In [1], a class of program schemes based around the usage of a stack was defined. A *program scheme* $\rho \in$ NPSS$_s$ involves a finite set $\{x_1, x_2, \ldots, x_k\}$ of *variables*, for some $k \geq 1$, and is over a signature τ. It consists of a finite sequence of *instructions* where each instruction is one of the following:

- an *assignment instruction* of the form x_i `:=` y, where $i \in \{1, 2, \ldots, k\}$ and where y is a variable from $\{x_1, x_2, \ldots, x_k\}$, a constant symbol of τ or one of the special constant symbols 0 and *max* (which do not appear in any signature);
- a *guess instruction* of the form `guess` x_i, where $i \in \{1, 2, \ldots, k\}$;
- a *while instruction* of the form `while` φ `do` α_1`;` α_2`; ...;` α_q`; od`, where φ is a quantifier-free formula over $\tau \cup \langle succ, 0, max \rangle$ whose free variables are from $\{x_1, x_2, \ldots, x_k\}$ and where each of $\alpha_1, \alpha_2, \ldots, \alpha_q$ is another instruction of one of the forms given here (note that there may be nested while instructions);
- a *push instruction* of the form `push` x_i, where $i \in \{1, 2, \ldots, k\}$;
- a *pop instruction* of the form x_i `:= pop`, where $i \in \{1, 2, \ldots, k\}$;

- an *accept* (resp. a *reject*) instruction `accept` (resp. `reject`).

A program scheme $\rho \in \mathrm{NPSS}_s$ over τ takes a τ-structure $\mathcal{A}$ as input. The program scheme ρ computes on $\mathcal{A}$ in the obvious way except that:

- the binary relation *succ* is taken to be any successor relation on $|\mathcal{A}|$, and 0 (resp. *max*) is the minimal (resp. maximal) element of this successor relation;
- initially, every variable takes the value 0;
- the pop and push instructions provide access to a stack which is initially empty;
- execution of the instruction `guess` x_i non-deterministically assigns an element of $|\mathcal{A}|$ to the variable x_i.

An input τ-structure $\mathcal{A}$, together with a chosen successor relation *succ*, is *accepted* by ρ if there is at least one computation of ρ on input $\mathcal{A}$ (and with regard to the chosen successor relation) leading to an accept instruction (if ever a pop of an empty stack is attempted in some computation then that computation is deemed rejecting). However, we only consider program schemes ρ of NPSS_s that are successor-invariant, where *successor-invariant* means that every input structure is such that it is either accepted no matter which successor relation is chosen or rejected no matter which successor relation is chosen. Such successor-invariant program schemes accept classes of structures closed under isomorphisms, *i.e.*, problems. Henceforth, we assume that all our program schemes are successor-invariant, and we equate (a class of) program schemes with the (class of) problems they accept.

Remark 1 In finite model theory, the way a 'built-in' successor relation is incorporated into a logic is exactly as we incorporate our successor relation into our program schemes. Essentially, having a built-in successor relation means that our data is *ordered* (as is the case for most standard models of computation). In order to work with *unordered* data, as one does in database theory, for example, one simply omits the inclusion of any such built-in successor relation in one's logic. We insist that our data is ordered as we are interested in the computational power of our devices in relation to other resource-bounded models of computation (such as Turing machines, which work on ordered data). The reader is referred to, for example, [5, 9, 12] for more on built-in (successor) relations.

Remark 2 In [1], the class of program schemes NPSS_s as defined above was actually called $\mathrm{NPSS}_s(1)$ and was the first class in an infinite hierarchy of classes of program schemes, the union of which was called NPSS_s. However, it was shown that (in the presence of a built-in successor relation) this infinite hierarchy collapses to the first class, $\mathrm{NPSS}_s(1)$, and that the resulting class of program schemes accepts exactly the class of polynomial-time solvable problems $\mathbf{P}$; so, we omit the suffix '(1)' for simplicity. There are also one or two purely cosmetic differences between the program schemes of $\mathrm{NPSS}_s(1)$ in [1] and the program schemes of NPSS_s in this paper that are of no relevance whatsoever.

We extend our class of program schemes NPSS_s to the class of program schemes NPSDS_s by allowing deep-push instructions:

- a *deep-push instruction* is of the form `dpush(y,z)`, where $\mathbf{y}$, the *index*, is a tuple of variables from $\{x_1, x_2, \ldots, x_k\}$ and constant symbols from $\tau \cup \langle 0, max \rangle$ (with repetitions allowed) and where z is a variable from $\{x_1, x_2, \ldots, x_k\}$ or a constant symbol from $\tau \cup \langle 0, max \rangle$

(we assume that the variables appearing in our program scheme are $x_1, x_2, \ldots, x_k$).

Suppose that ρ is some program scheme with deep-push instructions and that $\mathcal{A}$ is input to ρ, with associated successor relation *succ* (and constants 0 and *max*). Suppose that `dpush(y,z)` is a deep-push instruction appearing in ρ, where $\mathbf{y}$ is an m-tuple. The m-tuples of $|\mathcal{A}|^m$ are ordered lexicographically as

$$(0, 0, \ldots, 0), (0, 0, \ldots, u_1), (0, 0, \ldots, u_2), \ldots, (max, max, \ldots, max),$$

where $0, u_1, u_2, \ldots, max$ is the linear order encoded within the successor relation *succ*. Associate these m-tuples with the integers $0, 1, \ldots, n^m - 1$, respectively. Denote the integer associated with the tuple $\mathbf{u}$ as $\sharp\mathbf{u}$, and denote the tuple of values associated with the integer $i \in \{0, 1, \ldots, n^m - 1\}$ as $i\sharp$. When the instruction `dpush(y,z)` is executed, the variables of $\mathbf{y}$ each hold a value from $|\mathcal{A}|$; so, $\sharp\mathbf{y} \in \{0, 1, \ldots, n^m - 1\}$. The instruction `dpush(y,z)` pushes the value of z to the stack location $\sharp\mathbf{y}$ (the *offset*) from the top of the stack, with the top of the stack being the location 0 from the top of the stack, the next location down being the location 1 from the top of the stack and so on; in particular, the deep-push overwrites the value in the stack location $\sharp\mathbf{y}$ with the value of z. Thus, the instruction has access to the top n^m locations of the stack for pushing *but not for popping*. Only pops of the top element of the stack are allowed (in the usual way). If ever a deep-push attempts to push a value to some location below the bottom location of the stack then that particular computation is deemed rejecting. As usual, we only ever work with successor-invariant program schemes of NPSDS_s.

Let us illustrate deep-pushes with an example. However, before we do so, let us introduce some syntactic sugar. First, we employ the usual if-then-else instructions directly, as such instructions can easily be constructed using while instructions. Second, in describing our program schemes we use a convenient shorthand in relation to the offsets and indices in deep-push instructions. According to our syntax, the offset of a deep-push instruction is given via a tuple of variables, the index, with the offset value calculated according to the successor relation accompanying an input structure. Rather than include routine 'house-keeping' code, we allow ourselves to describe offset values in an integer-style. For example, suppose that we wished to use an offset of $2n - 2$ in a deep-push instruction (where n is the size of the input structure). Strictly speaking, we should use a pair of variables, (x_1, x_2), and include the following portion of code:

```
x1 := 0;
x2 := 0;
guess z;
while ¬succ(x1, z) do
   guess z;
od;
x1 := z;
guess z;
```

```
while ¬succ(z, max) do
   guess z;
od;
x_2 := z;
dpush((x_1,x_2),y);
```

However, we abbreviate the above with the instruction:

```
dpush((2n - 2)♯,y);
```

In general, if i is any positive integer then we write $i\sharp$ to denote a tuple of variables (of the required length) encoding the offset of value i. All of the integer offsets appearing in our program schemes are such that they can easily be computed with an appropriate portion of code. Thus, for example, the instruction:

```
dpush(((n - ♯count) + ♯y)♯,y);
```

is shorthand for a portion of code which builds values for a tuple of 2 variables which encodes an integer equal to n minus the integer currently encoded by the current value of the variable *count* plus the integer encoded by the current value of the variable y. We also use the above shorthand in assignments, *e.g.*, $(count, y)$ `:= (♯`$(count, y)$ `+ 1)♯`. Furthermore, we abuse our notation by writing, for example, *count* `:=` $2n\sharp$ or `while ♯`x `<` $2n-1$ `do`, where what we should really write is $(count_1, count_2)$ `:=` $2n\sharp$ and `while ♯`(x_1, x_2) `<` $2n-1$. (As can be seen, we often use names for variables different to $x_1, x_2, \ldots$)

Example 3 Consider the following program scheme ρ of NPSDS$_s$ over the signature $\tau = \langle E, C, D\rangle$, where E is a binary relation symbol and C and D are constant symbols (so, an input structure can be considered as a digraph with two specified vertices):

```
1    guess w;                  ** push some 0's onto the stack **
2    while w = 0 do
3       push 0;
4       guess w;
5    od;
6    count := 0;               ** count counts the number of guessed **
7    dpush(count,C);           ** vertices in a path making sure that **
8    dpush((n + ♯C)♯,max);     ** n are guessed and registered       **
9    while count ≠ max do
10      count := (♯count + 1)♯;
11      guess x;
12      dpush(count,x);
13      dpush((n + ♯x)♯,max);
14   od;
15   count := 0;               ** count counts the number of vertices **
16   x := pop;                 ** popped when checking the validity   **
17   while count ≠ max do      ** of the guessed path                 **
18      count := (♯count + 1)♯;
19      y := pop;
20      if ¬E(x,y) then reject; fi;
```

```
21      x := y;
22    od;
23    if x ≠ D then reject; fi;
24    count := 0;                ** count counts the number of vertex **
25    w := pop;                  ** registrations checked so far      **
26    if w ≠ max then reject; fi;
27    while count ≠ max do
28      count := (♯count + 1)♯;
29      w := pop;
30      if w ≠ max then reject; fi;
31    od;
32    accept;
```

Essentially, on an input digraph of size n, ρ begins by building a stack of 0s, in lines 1-5. In lines 6-14, a path of n vertices is guessed, starting with vertex C, and (using deep-pushes) these vertices are stored in order in the top n locations in the stack (with C in the top location, the next vertex in the location with offset 1, and so on). When a vertex u is guessed, this is registered by deep-pushing max to the stack location with offset $n+\sharp u$. After this guessing phase, in lines 15-23 the path is popped from the stack and checked as to whether it is a valid path (ending in D). If so then the registrations are then checked, in lines 24-32, to confirm that all vertices appear once on the path. Hence, an input structure is accepted by ρ if, and only if, $C \neq D$ and there is a Hamiltonian path from vertex C to vertex D. Note that the program scheme ρ is successor-invariant. □

Let us close this section with a remark. Given our wish to augment basic program schemes with a deep pushdown stack, we need some mechanism for our program schemes to access locations within the stack. Working with ordered data and using the built-in successor relation and tuples of variables for constructing numeric offsets gives us this mechanism. This naturally gives us the limitation that deep-pushes can only be 'polynomially-deep'. An alternative would be to drop the built-in successor relation and introduce an additional domain of numeric values and have variables of two sorts. We have chosen to work on ordered data as this is the norm in descriptive complexity.

3 Sub-classes of program schemes

We begin by showing that restricting the values pushed by push and deep-push instructions to 0 and max does not limit the problems accepted by our program schemes; we call such program schemes *boolean* program schemes. Let us denote the class of program schemes where push instructions must be of the form `push` 0 or `push` max and where deep-push instructions must be of the form `dpush(y,0)` or `dpush(y,`max`)` by NPSDS_s^b (with the superscript b reflecting the boolean nature of such program schemes).

Proposition 4 Any problem accepted by a program scheme of NPSDS_s can be accepted by a program scheme of NPSDS_s^b.

Proof Let ρ be a program scheme of NPSDS$_s$. Essentially, we simulate the stack in a computation of ρ on some input of size n with a stack in a computation of a new program scheme ρ' so that n locations of the new stack correspond to 1 location of the old stack. The values of the n stack locations of the new stack corresponding to the 1 stack location of the old stack are such that all are 0 except for one which is max and the stack location that is max is the ith in the batch of n stack locations if, and only if, the value of the corresponding old stack location is u_i, where u_i is the ith smallest element in the ordering given by the successor relation *succ*. The instructions of ρ are adapted accordingly to obtain our program scheme ρ'.

So, for example, `push` x in ρ is simulated by:

```
x' := 0;
while x' ≠ x do
   push 0;
   x' := (♯x' + 1)♯;
od;
push max;
while x' ≠ max do
   push 0;
   x' := (♯x' + 1)♯;
od;
```

The instruction x `:= pop` in ρ is simulated by:

```
x' := 0;
x := pop;
while x ≠ max do
   x' := (♯x' + 1)♯;
   x := pop;
od;
x := x';
while x' ≠ max do
   x' := (♯x' + 1)♯;
   y := pop;
od;
```

The instruction `dpush(y,`x`)` in ρ is simulated by:

```
y' := 0;
while y' ≠ x do
   dpush((y,y'),0);
   y' := (♯y' + 1)♯;
od;
dpush((y,y'),max);
while y' ≠ max do
   dpush((y,y'),0);
   y' := (♯y' + 1)♯;
od;
```

Note how the length of the index tuple has increased by 1 in our simulation of the instruction `dpush(y,`x`)`. All other instructions in ρ remain unaltered. It is clear that the resulting program scheme ρ' is as required. □

We also restrict the syntax of program schemes of NPSDS$_s$ (and NPSDS$_s^b$) by limiting the usage of pops, pushes and deep-pushes in phases of a program scheme. A *batch* of instructions is a well-formed sequence of instructions so that if one of these instructions is a while-do instruction (resp. if instruction) then the corresponding while-od instruction (resp. fi instruction) must also appear in the batch. Let ρ be a program scheme of NPSDS$_s$. If ρ can be written (as a concatenation of instructions) as a batch of instructions ρ_0, followed by a batch of instructions ρ_1, and so on, finally ending with a batch of instructions ρ_k, then we write $\rho = (\rho_0, \rho_1, \ldots, \rho_k)$. The allowed usage of pops, pushes and deep-pushes in any batch of instructions is signalled as follows:

- if pops are allowed (resp. disallowed) then we signal this with o (resp. $\overline{o}$);
- if pushes are allowed (resp. disallowed) then we signal this with u (resp. $\overline{u}$);
- if deep-pushes are allowed (resp. disallowed) then we signal this with d (resp. $\overline{d}$).

Thus, if pops and pushes are allowed in some batch of instructions ρ_i, but not deep-pushes, then we write $\rho_i \in \text{NPSDS}_s(ou\overline{d})$. We adapt our notation to situations where a program scheme is the concatenation of a sequence of batches of instructions by detailing the allowed usage of pops, pushes and deep-pushes in each batch by a sequence of parameters. So, for example, if $\rho = (\rho_0, \rho_1, \rho_2)$ where $\rho_0 \in \text{NPSDS}_s(ou\overline{d})$, $\rho_1 \in \text{NPSDS}_s(\overline{o}u\overline{d})$ and $\rho_2 \in \text{NPSDS}_s(o\overline{u}d)$, then we write $\rho \in \text{NPSDS}_s(ou\overline{d}, \overline{o}u\overline{d}, o\overline{u}d)$. The above applies equally to program schemes of NPSDS$_s^b$.

Note that the proof of Proposition 4 does not alter the interleaving of pops, pushes and deep-pushes in the simulating program scheme. So, for example, the problem accepted by some program scheme of NPSDS$_s(ou\overline{d}, \overline{o}u\overline{d}, o\overline{u}d)$ can be accepted by some program scheme of NPSDS$_s^b(ou\overline{d}, \overline{o}u\overline{d}, o\overline{u}d)$.

4 Capturing NP

In this section, we define a sub-class of program schemes of NPSDS$_s$ that captures exactly the complexity class **NP**.

Proposition 5 Every program scheme $\rho = (\rho_0, \rho_1, \rho_2)$ in NPSDS$_s(ou\overline{d}, \overline{ou}d, ou\overline{d})$ accepts a problem in **NP**.

Proof Let $\rho = (\rho_0, \rho_1, \rho_2)$ be a program scheme of NPSDS$_s(ou\overline{d}, \overline{ou}d, ou\overline{d})$ over τ. Thus, ρ_0 and ρ_2 do not contain deep-pushes, and ρ_1 contains neither pushes nor pops. We begin by deriving a program scheme ρ' of NPSS$_s$ from ρ. Essentially, the program scheme ρ' will simulate ρ except that we shall replace the computation of ρ_1 with an 'oracle' and amend ρ_2 accordingly. However, because of subtleties with our simulation (as we shall detail soon), we also need to amend ρ_0 too.

Let us begin by looking at the structure of a computation of ρ on some input structure of size n. The program scheme ρ_0 builds, using pops and pushes, a stack, the height of which will, in general, vary until ρ_0 terminates. At this point, the program scheme ρ_1 begins computing and goes on to, in general, perform some deep-pushes, with an offset of at most $n^k - 1$, for some k. Note that the height of the stack does not change throughout the computation of ρ_1. When ρ_1 terminates, the program scheme ρ_2 begins computing and, due to pops and pushes, the height of the stack, in general, varies.

What we intend to do is to 'remove' the computation of ρ_1 from ρ and replace it with an 'oracle' which supplies both the locations to which deep-pushes are made and the values pushed, and also the values of the variables of ρ on termination of ρ_1. We will then amend ρ_2 so that when a pop is made of an element from the stack which might have potentially been altered by a deep-push, we consult our oracle to see whether there was indeed a deep-push to this location. If there was then we take the popped value as the value pushed, otherwise we leave the popped value as whatever was popped. We shall return to how we obtain our oracle later.

One can immediately see that we need to amend ρ_2 so that we keep track of the n^k stack locations into which deep-pushes might have occurred. However, it may be the case that ρ_2 amends the stack through popping and pushing, and this complicates matters. The obvious solution is for us to maintain a 'counter' which keeps track of the highest location in the (current) stack within which a deep-push might have occurred (as an offset from the top of the current stack), and also the offset from this location covering all stack locations (downwards) within which a deep-push might have occurred; that is, two offsets detailing the portion of the current stack into which any deep-pushes have been made. However, any counter can have a maximum value of n^m, for some fixed m, and (potentially) ρ_2 might increase the height of the stack by an exponential (in n) number. Thus, it does not appear that we can use a counter or counters to keep track of the portion of the stack as described above. The situation can be visualized as in Fig. 1, where the evolution of the stack during the execution of ρ_2 is depicted, with the shaded portion of the stack corresponding to the locations within which a deep-push might have occurred. In the right-hand stack, if we keep a counter telling us how far the top shaded location (in which the value v_r resides) is from the top of the stack then this value could be greater than n^m (even exponential in n).

We get round the above difficulty by changing completely how data is stored on the stack. First, we rewrite ρ so that every push or pop is replaced by a pair of pushes or pops, and every deep-push is replaced by a single deep-push. In particular: we replace the instruction `push` x_i with the two instructions `push` x_i`;` `push` 0; we replace the deep-push instruction `dpush(x,`y`)` with the instruction `dpush(x'`$,y$`)`, where $\sharp\mathtt{x}' = 2 \cdot \sharp\mathtt{x} + 1$; and we replace the instruction x `:= pop` with the two instructions x `:= pop;` x `:= pop`. Note that we need to increase the index of deep-pushes to cater for the increased size of stack. So, it appears that we are wasting stack locations; however, we will use the stack locations in which (seemingly redundant) 0's have been pushed as locations within which markers are held (where a marker is the value max). Denote this amended version of $\rho = (\rho_0, \rho_1, \rho_2)$ by $\tilde{\rho} = (\tilde{\rho}_0, \tilde{\rho}_1, \tilde{\rho}_2)$.

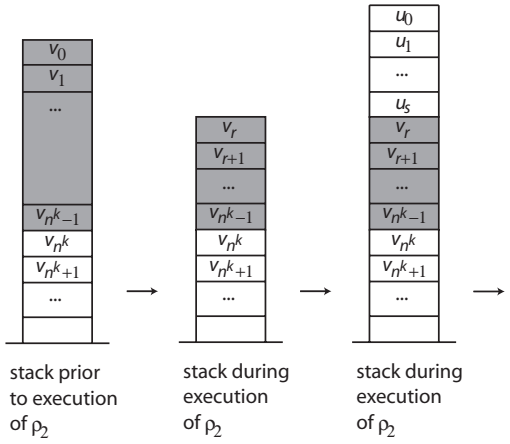

Figure 1. The evolution of the stack.

Now let us return to how we amend $\tilde{\rho}_2$ so that we can ascertain whether a stack location is a location within which a deep-push might have occurred. The first thing our amended version of $\tilde{\rho}_2$, call it ρ_2', does is to pop the top element from the stack and then push onto the stack the value max. This marks the 'top' of the portion of the stack within which a deep-push might have a occurred (that is, with reference to Fig. 1, the highest shaded location). We also set a counter, call it the tuple of variables $\mathbf{y}$, so that $\sharp\mathbf{y} = 0$. This counter will denote the distance of the current marked stack location from the initial marked stack location; of course, as some of the values in these 'shaded' locations are popped from the stack during an execution of ρ_2', the value of $\sharp\mathbf{y}$ will increase. So, again with reference to the right-hand stack of Fig. 1, at this point $\sharp\mathbf{y}$ will be r to denote that r elements of the original 'shaded' portion of the stack have been popped (as can be seen from the diagram, other elements have since been pushed onto the stack). Note that the value of $\sharp\mathbf{y}$ should be between 0 and $2n^k - 2$, and so we never have any difficulties with this counter becoming too large (of course, if $\sharp\mathbf{y}$ assumes a value greater than $2n^k - 2$ then this means that we have popped all values in the original 'shaded' locations from the stack into which a deep-push might have occurred and we can stop worrying about deep-push information).

We also amend ρ_2' so that we replace every pair of pops x_i `:= pop;` x_i `:= pop` (recall, all pops come in pairs in $\tilde{\rho}_2$, apart from the initial pop added above) with the following sequence of instructions:

```
z := pop;
x_i := pop;                                                    (*)
if z = max then
   if ♯y ≠ 2n^k then
      z := pop;
      push max;
      y := (♯y + 2)♯;
   fi;
fi;
```

It should be clear that ρ_2' keeps track of the current stack locations within which a deep-push might have occurred (during an execution of $\tilde{\rho}_1$).

Now for our 'oracle' containing information as regards the computation of $\tilde{\rho}_1$. Let R (resp. T) be a relation symbol (not in τ) of arity (resp. one plus) the maximal index, m', of any deep-push instruction in $\tilde{\rho}$ (recall that this index is one more than the maximal index of any deep-push instruction in ρ). In every (new) portion of code corresponding to a pair of pops, as laid out in the previous paragraph, replace the pop in line $(*)$ with the instructions:

```
x_i := pop;
if ♯y ≠ 2n^k then
   w := (♯y + 1)♯;
   if R(w) then
      guess z;
      if T(w, z) holds then
         x_i := z;
      else
         reject;
      fi;
   fi;
fi;
```

The intention is that R and T will detail the deep-push locations and the values pushed in a specific computation of $\tilde{\rho}_1$ via: for every $\mathbf{u} \in |\mathcal{A}|^{m'+1}$, $R(\mathbf{u})$ holds if, and only if, a deep-push occurs to the 'shaded' location indexed by $\mathbf{u}$, and if there is a deep-push to location $\mathbf{u}$ then $T(\mathbf{u}, v)$ holds if, and only if, the final value deep-pushed to this location is v (note that there may be multiple deep-pushes to some stack location in a computation).

Now amend the program scheme $\tilde{\rho}_1$ so that all deep-push instructions are simply omitted, and denote this amended program scheme by ρ'_1. Note that $\rho'_1 \in \mathrm{NPSS}_s$; in fact, $\rho'_1 \in \mathrm{NPS}_s$, the sub-class of program schemes with no stack. Let $\mathcal{A}$ be any τ-structure that is accepted by $\tilde{\rho}$ and suppose that the relations $R^{\mathcal{A}}$ and $T^{\mathcal{A}}$ are such that they detail *exactly* the locations where the deep-pushes were made and the values deep-pushed (during $\tilde{\rho}_1$) during a particular accepting execution of $\tilde{\rho}$. Then $(\mathcal{A}, R^{\mathcal{A}}, T^{\mathcal{A}})$ is accepted by the program scheme $(\tilde{\rho}_0, \rho'_1, \rho'_2)$.

Let us go further and omit the computation of ρ'_1. In more detail, let $\mathbf{x}$ denote the t-tuple of all variables appearing in $\tilde{\rho}_0$ and ρ'_1, and let $\mathbf{c} = (c_1, c_2, \ldots, c_t)$ and $\mathbf{d} = (d_1, d_2, \ldots, d_t)$ be t-tuples of new and distinct constant symbols. Define ρ' to be the following program scheme of NPSS_s:

```
ρ̃_0
if x ≠ c then
   reject;
fi;
x := d;
ρ'_2
```

Let $\mathcal{A}$ be any τ-structure that is accepted by $\tilde{\rho}$ and suppose that in a particular accepting computation: $\mathbf{c}$ details the values of the variables of $\mathbf{x}$ immediately after termination of $\tilde{\rho}_0$; $\mathbf{d}$ details the values of the variables of $\mathbf{x}$ immediately after termination of ρ'_1; and the relations $R^{\mathcal{A}}$ and $T^{\mathcal{A}}$ are such that they detail exactly the

locations where the deep-pushes were made and values deep-pushed (during $\tilde{\rho}_1$) during the execution of $\tilde{\rho}$. Then $(\mathcal{A}, R^{\mathcal{A}}, T^{\mathcal{A}}, \mathbf{c}, \mathbf{d})$ is accepted by the program scheme ρ'.

Consider the following algorithm, where the input is a τ-structure $\mathcal{A}$:

```
guess a successor relation succ;
guess relations R^A and T^A over |A| and tuples of constants
c and d from |A|;
if (A, R^A, T^A, c, d) ⊭ ρ' then
   reject;
else
   simulate ρ'_1 on A with the variables starting with the values
   given by c and store all stack locations to which a deep-push
   was made and the final values deep-pushed using R_0^A and T_0^A;
   if the execution of ρ'_1 does not end with the values of the
   variables given by d then
      reject;
   else
      if R_0^A ≠ R^A or T_0^A ≠ T^A then
         reject;
      else
         accept;
```

We clearly have that $\mathcal{A} \models \tilde{\rho}$ if, and only if, $\mathcal{A}$ is accepted by the above algorithm. Moreover, the above algorithm is a non-deterministic polynomial-time algorithm, as $\rho' \in \mathrm{NPSS}_s$ and $\rho'_1 \in \mathrm{NPS}_s$; hence, the result follows. □

Denote by NPSDS_s^{+b} the sub-class of program schemes of NPSDS_s^b where any instruction involving a deep-push must be of the form `dpush(y,`max`)`; that is, only the value max can be deep-pushed to a stack location. We can use our program scheme for the problem HP in Example 3 to show that the class of program schemes $\mathrm{NPSDS}_s^{+b}(ou\overline{d}, \overline{ou}d, ou\overline{d})$ actually contains **NP**.

Proposition 6 For every problem Ω in **NP**, there exists a program scheme of $\mathrm{NPSDS}_s^{+b}(ou\overline{d}, \overline{ou}d, ou\overline{d})$ accepting Ω.

Proof Recall from [4, 18] that any problem in **NP**, over the signature τ, say, can be described by a sentence Φ of the form:

$$\mathrm{HP}[\lambda \mathbf{x}, \mathbf{y} \varphi(\mathbf{x}, \mathbf{y})](\mathbf{0}, \mathbf{max}),$$

where: $\mathbf{x}$ and $\mathbf{y}$ are m-tuples of variables, for some $m \geq 1$, with all variables distinct; φ is a quantifier-free formula over $\tau \cup \langle succ, 0, max \rangle$; and $\mathbf{0}$ (resp. $\mathbf{max}$) is the constant symbol 0 (resp. max) repeated m times. A τ-structure $\mathcal{A}$ *satisfies* Φ if, when we build the digraph $\varphi(\mathcal{A})$ with vertex set $|\mathcal{A}|^m$ and where there is an edge $(\mathbf{u}, \mathbf{v})$ if, and only if, $\varphi^{\mathcal{A}}(\mathbf{u}, \mathbf{v})$ holds, there is a Hamiltonian path in $\varphi(\mathcal{A})$ from the vertex $\mathbf{0}$ to the vertex $\mathbf{max}$ (we assume that there is a built-in successor relation).

Given our program scheme ρ in Example 3, we begin by amending ρ so that it is a program scheme of $\mathrm{NPSDS}_s^{+b}(ou\overline{d}, \overline{ou}d, ou\overline{d})$. This means dealing with the instructions in lines 7 and 12 where a value different from max might be deep-pushed. Replace the instruction in line 7 with:

```
z := 0;
while z ≠ C do
   z := (♯z + 1)♯;
od;
dpush((count,z),max);
```

and the instruction in line 12 with identical code except that C is replaced by x. Also, replace the instruction in line 8 with:

```
dpush((n^2 + ♯C)♯,max);
```

and the instruction in line 13 with identical code except that C is replaced by x. Replace the instruction in line 16 with:

```
z := 0;
w := pop;
while w ≠ max do
   z := (♯z + 1)♯;
   w := pop;
od;
x := z;
while z ≠ max do
   z := (♯z + 1)♯;
   w := pop;
od;
```

and the instruction in line 19 with analogous code. The resulting program scheme ρ' of $\mathrm{NPSDS}_s^{+b}(ou\overline{d}, \overline{ou}d, ou\overline{d})$ is:

```
1     guess w;
2     while w = 0 do
3        push 0;
4        guess w;
5     od;
6     count := 0;
7.1   z := 0;
7.2   while z ≠ C do
7.3      z := (♯z + 1)♯;
7.4   od;
7.5   dpush((count,z),max);
8.1   dpush((n^2 + ♯C)♯,max);
9     while count ≠ max do
10       count := (♯count + 1)♯;
11       guess x;
12.1     z := 0;
12.2     while z ≠ x do
12.3        z := (♯z + 1)♯;
12.4     od;
12.5     dpush((count,z),max);
```

```
13.1    dpush((n^2 + ♯x)♯,max);
14    od;
15    count := 0;
16.1  z := 0;
16.2  w := pop;
16.3  while w ≠ max do
16.4     z := (♯z + 1)♯;
16.5     w := pop;
16.6  od;
16.7  x := z;
16.8  while z ≠ max do
16.9     z := (♯z + 1)♯;
16.a     w := pop;
16.b  od;
17    while count ≠ max do
18       count := (♯count + 1)♯;
19.1     z := 0;
19.2     w := pop;
19.3     while w ≠ max do
19.4        z := (♯z + 1)♯;
19.5        w := pop;
19.6     od;
19.7     y := z;
19.8     while z ≠ max do
19.9        z := (♯z + 1)♯;
19.a        w := pop;
19.b     od;
20       if ¬E(x, y) then reject; fi;
21       x := y;
22    od;
23    if x ≠ D then reject; fi;
24    count := 0;
25    w := pop;
26    if w ≠ max then reject; fi;
27    while count ≠ max do
28       count := (♯count + 1)♯;
29       w := pop;
30       if w ≠ max then reject; fi;
31    od;
32    accept;
```

and ρ' accepts the same problem as that accepted by ρ.

We now amend ρ' so that it accepts the problem described by the sentence Φ. Essentially, all we need to do is to: replace the variables *count*, z, x and y with the m-tuples of variables **count**, **z**, **x** and **y**, respectively (we leave the variable w as it is); replace the n^2 in lines 8.1 and 13.1 with n^{m+1}; replace the constant symbol C with the m-tuple **0**, and the constant symbol D with the m-tuple **max**; and replace the test $E(x, y)$ with the test $\varphi(\mathbf{x}, \mathbf{y})$ (of course, an instruction such as *count* := 0

becomes $count_1$:= 0; $count_2$:= 0; ...; $count_m$:= 0, and so on). Having done this, the new program scheme accepts the problem described by Φ and the result follows. □

Propositions 5 and 6 immediately yield the following corollary.

Corollary 7 $\mathrm{NPSDS}_s^{+b}(ou\overline{d}, \overline{ou}d, ou\overline{d}) = \mathrm{NPSDS}_s(ou\overline{d}, \overline{ou}d, ou\overline{d}) = \mathbf{NP}$.

5 Capturing PSPACE

In this section, we define a sub-class of program schemes of NPSDS_s capturing the complexity class **PSPACE**. However, before we do this let us demonstrate the power of program schemes of NPSDS_s and exhibit a program scheme accepting the complement of the problem in Example 3.

Example 8 Let $\tau = \langle E, C, D\rangle$, with E a relation symbol of arity 2 and C and D constant symbols. The problem co-HP is defined as

$$\{\mathcal{A} \ : \ \mathcal{A} \text{ is a } \tau\text{-structure and there is not a Hamiltonian path in the digraph with edges given by the relation } E^{\mathcal{A}} \text{ from vertex } C^{\mathcal{A}} \text{ to vertex } D^{\mathcal{A}}\}.$$

We shall exhibit a program scheme ρ of NPSDS_s accepting co-HP.

We outline what the program scheme ρ that accepts co-HP does, in an intuitive sense, before presenting the actual program scheme in detail. The program scheme ρ begins by simply non-deterministically building a 'sufficiently tall' stack of 0's (in actuality, this stack should have height at least $2n^n$). The top n elements of the stack (the 'top batch') are regarded as the current potential Hamiltonian path of vertices in the input digraph; initially, of course, this path is $0, 0, \ldots, 0$. The next n elements of the stack (the 'middle batch') are regarded as work-space that will enable us to verify that there are no repeated vertices in the current path. The next n elements of the stack (the 'lower batch') will be such that they will contain the lexicographically-next potential Hamiltonian path.

The top element (of the top batch) is popped from the stack and, using a deep-push, the name of this vertex, u, say, is registered in location $\sharp u$ of the middle batch by writing max (note that the top batch of elements now consists of $n-1$ elements). A check is also made to verify that u is, in fact, the constant C. We iteratively pop elements from the stack and verify that the current path is indeed a path in our input digraph, registering these elements, as above, as we proceed. Also, alongside our checking of the current potential Hamiltonian path, we build the lexicographically-next path in the lower batch of stack elements. Essentially, as we pop the vertices of the current path from the top batch of the stack, we look for the location *place* that holds an element that is not equal to max but where every location in the top batch lower than this location *place* holds an element equal to max; the element in this location *place* is to be incremented by 1 and all elements in the top-batch at lower locations are to be set to 0 to get the lexicographically-next potential Hamiltonian path. This is done, using deep-pushes, so that the lexicographically-next path appears in the lower batch. Having popped all the vertices of the current path from the stack, we verify that the last element popped was D and we use our registrations to

verify that the path is indeed Hamiltonian. If not then we proceed as above except that what was the lower batch is now the top batch and we are working with the lexicographically-next potential Hamiltonian path, as our current path.

The program scheme ρ implementing the above procedure follows.

```
1     guess x;                    ** fill the stack with 0's **
2     while x = 0 do
3        push x;
4        guess x;
5     od;
6     done := 0;                  ** done = 0 denotes we haven't finished **
7     while done = 0 do           ** checking all paths                    **
8        done := max;
9        bad_path := 0;           ** bad_path = 0 means that the path is still **
10       x := pop;                ** potentially good                      **
11       count := 1♯;             ** count counts the vertices popped **
12       dpush(((n - ♯count) + ♯x)♯,max);        ** register x in the **
13       if x ≠ max then                         ** middle batch     **
14          store := x;           ** remember the last location whose contents **
15          place := (2n - 1)♯;   ** are different from max                 **
16          done := 0;
17       fi;
18       if x ≠ C then            ** if the first vertex ≠ C then the path is bad **
19          bad_path := max;
20       fi;
21       dpush((2n - 1)♯,x);      ** build the next path in the lower batch **
22       while ♯count < n do      ** iteratively pop the path vertices **
23          y := pop;
24          count := (♯count + 1)♯;
25          if ¬E(x,y) then       ** check that there is a path-edge from x to y **
26             bad_path := max;
27          fi;
28          dpush(((n - ♯count) + ♯y)♯,max);        ** register y **
29          if y ≠ max then
30             store := y;              ** remember the last location whose **
31             place := (2n - 1)♯;      ** contents are different from max  **
32             done := 0;
33          else
34             if done = 0 then
35                place := (♯place - 1)♯;
36             fi;
37          fi;
38          dpush((2n - 1)♯,y);         ** continue building the next path **
39          x := y;
40       od;
41       if done = 0 then
42          dpush(place,(♯store + 1)♯);        ** perform the final stage of **
43          place := (♯place + 1)♯;            ** building the next path     **
```

```
44          while ♯place < 2n do
45             dpush(place,0);
46             place := (♯place + 1)♯;
47          od;
48          if x ≠ D then                    ** check that the final vertex **
49             bad_path := max;              ** is D                        **
50          fi;
51          count := 0;
52          while ♯count < n do              ** check that the path is **
53             x := pop;                     ** indeed Hamiltonian    **
54             if x ≠ max then
55                bad_path := max;
56             fi;
57             count := (♯count + 1)♯;
58          od;
59          if bad_path := 0 then            ** a Hamiltonian path has **
60             reject;                       ** been found so reject    **
61          fi;
62       else
63          accept;              ** we have checked all paths and no **
64       fi;                     ** Hamiltonian path has been found **
65    od;
```

Given our intuitive description above, it should be clear that ρ is an implementation of our algorithm. The only further remark we have is that deep-pushes are parameterized by an offset from the top of the stack and so this offset needs to be continually calculated. The offset as regards the registration of vertices found so far (in the middle batch, as undertaken in lines 12 and 28) is calculated by remembering the number of vertices currently popped from the top batch (namely the value $\sharp count$). The offset as regards the building of the lexicographically-next path (in the lower batch, as undertaken in lines 21 and 38) is $2n-1$. The final phase of building the lexicographically-next path, where a path-location is incremented by 1 and all subsequent path-locations set at 0, is undertaken in lines 42 and 45. □

Having demonstrated the power of program schemes of NPSDS$_s$, we now turn to capturing the complexity class **PSPACE**.

Proposition 9 Any polynomial-space Turing machine can be simulated by a program scheme of NPSDS$_s(\overline{o}u\overline{d}, o\overline{u}d)$ so that the stack of the program scheme encodes the evolution of the work-tape of the Turing machine. Thus, **PSPACE** $\subseteq$ NPSDS$_s(\overline{o}u\overline{d}, o\overline{u}d)$.

Proof Let M be a Turing machine which uses n^k space (we assume that all input strings are of length at least 2) and which accepts the problem Ω over the signature τ (any τ-structure is encoded as a string, by assuming some successor relation on the domain of the structure, and M accepts exactly those strings encoding structures of Ω). We may assume that M has a one-way work-tape that is infinite to the right, that the input string initially appears in cells $0, 1, \ldots, n-1$ of the work-tape, and

that the only work-tape symbols are 0, 1 and b (b is the blank symbol; it is always obvious as to whether we are talking about the symbol 0 of the Turing machine or the constant symbol 0 of a program scheme). We now describe a program scheme ρ of NPSDS$_s(\overline{o}u\overline{d}, o\overline{u}d)$ which simulates M and thus accepts Ω.

The program scheme ρ begins by non-deterministically pushing 0's onto the stack. Next, the input string corresponding to the input structure $\mathcal{A}$ and the given successor relation *succ* is pushed onto the stack but in reverse order; that is, reading the stack from the top element downwards is akin to reading the tape of the Turing machine M as it stands initially and from left to right. However, rather than push the input string onto the stack verbatim, we encode this input string as follows: if a bit of the input string is a 1 then we push two max's onto the stack followed by a 0; and if a bit of the input string is a 0 then we push a max then a 0 then a 0 onto the stack. The only exception is that if the first bit of the input string is 1 then we push three max's onto the stack, and if the first bit of the input string is a 0 then we push a max then a 0 then a max onto the stack. So, for example, if the input string is 10110 then the top of the stack, from the top location downwards, will read:

$$max, max, max, 0, 0, max, 0, max, max, 0, max, max, 0, 0, max.$$

The top $3n^k$ elements of the stack encode the work-tape of M such that for every $i = 0, 1, \ldots, n^k - 1$:

- the location $3i$ from the top of the stack holds max if, and only if, the tape-head of M is pointing at cell i of the work-tape;
- the locations $3i+1$ and $3i+2$ from the top of the stack hold (max, max) (resp. $(0, max)$, $(0, 0)$) if cell i of the work-tape holds the symbol 1 (resp. 0, b).

The initial state of M is encoded using a tuple of variables in ρ.

In general, a move of M is simulated within ρ by popping the top $3n^k$ elements from the stack (which always encode the contents of M's work-tape, as above, prior to the simulation of a move) and, using deep-pushes, copying them to the next $3n^k$ elements of the stack, except that the move of M is registered in the new encoding of the work-tape appropriately. Note that the move can easily be simulated (using the variables of ρ) as the cell at which the tape-head is currently pointing can always be found (as the stack is being popped) and the move of M can be easily registered in the new description of the work-tape (no matter whether the tape-head moves left, moves right or stays stationary). It should be clear that the program scheme ρ accepts the input structure if, and only if, the corresponding input string (where the successor relation is taken as *succ*) is accepted by M. □

In fact, more can be said. It should be clear that the proof of Proposition 9 is such that the program scheme ρ constructed can actually be taken to be a program scheme of NPSDS$_s^{+b}(\overline{o}u\overline{d}, o\overline{u}d)$. Thus, we obtain the following strengthening of Proposition 9.

Corollary 10 Any polynomial-space Turing machine can be simulated by a program scheme of NPSDS$_s^{+b}(\overline{o}u\overline{d}, o\overline{u}d)$ so that the stack of the program scheme encodes the evolution of the work-tape of the Turing machine. Thus, **PSPACE** $\subseteq$ NPSDS$_s^{+b}(\overline{o}u\overline{d}, o\overline{u}d)$.

Before we proceed, we need some definitions.

Definition 11 Let ρ be a program scheme of NPSDS$_s$ and let $\mathcal{A}$ be an input structure to ρ. An *ID* of ρ on input $\mathcal{A}$ is a tuple α consisting of values (from $|\mathcal{A}|$) for the variables of ρ together with the next instruction to be executed. A *configuration* of ρ on input $\mathcal{A}$ is an ordered pair, written $[\alpha, \mathbf{s}]$, the first component of which, α, is an ID and the second component of which, $\mathbf{s}$, is a complete description of the stack.

Any computation of ρ on input $\mathcal{A}$ can be described as a sequence of configurations, with each of these configurations having an associated ID. Of course, some IDs and some configurations might not be achievable in any computation of ρ on input $\mathcal{A}$.

Proposition 12 Every program scheme ρ in NPSDS$_s^{+b}(u\overline{o}\overline{d}, o\overline{u}d)$ accepts a problem in **PSPACE**.

Proof Let $\rho = (\rho_0, \rho_1) \in$ NPSDS$_s^{+b}(\overline{o}u\overline{d}, o\overline{u}d)$. Suppose that on some input structure $\mathcal{A}$ of size n, ρ_1 is such that all deep-pushes occur with index at most k. We shall simulate the computation of ρ with a Turing machine M. By amending ρ similarly to as was done in the proof of Proposition 5, we may clearly assume that every computation of ρ_0 pushes at least n^k elements onto the stack.

Any ID of ρ_0 on input $\mathcal{A}$ can be stored on M's work-tape using $O(\log(n))$ tape-cells. Moreover, given any two IDs, say ID$_a$ and ID$_b$, and an element $u \in |\mathcal{A}|$, M can clearly decide whether there is a computation of ρ_0 on input $\mathcal{A}$ from the configuration $[\mathrm{ID}_a, \epsilon]$, *i.e.*, the ID ID$_a$ together with an empty stack, (resp. from the configuration $[\mathrm{ID}_a, (u)]$, *i.e.*, the ID ID$_a$ together with the stack consisting solely of the element u) to a configuration whose ID is ID$_b$; this computation can be done by M non-deterministically using $O(\log(n))$ space.

The computation of M begins with M guessing a sequence:

$$\mathrm{ID}_{n^k}, u_{n^k-1}, \mathrm{ID}_{n^k-1}, \ldots, u_1, \mathrm{ID}_1, u_0, \mathrm{ID}_0$$

where each ID$_i$ is an ID of ρ_0 on input $\mathcal{A}$ and where each $u_i \in |\mathcal{A}|$, with M then verifying that:

- for every $i \in \{1, 2, \ldots, n^k\}$, there is a computation of ρ_0 on input $\mathcal{A}$ from configuration $[\mathrm{ID}_i, \epsilon]$ to configuration $[\mathrm{ID}_{i-1}, (u_{i-1})]$;
- for every $i \in \{1, 2, \ldots, n^k\}$, the current instruction encoded within ID$_i$ is a push and the element pushed onto the stack is u_{i-1};
- ID$_0$ is a terminating ID of ρ_0; that is, an ID where the next instruction to be executed is the fist instruction of ρ_1.

Note that should this verification succeed, there is a terminating computation of ρ_0 on input $\mathcal{A}$ starting in configuration ID$_{n^k}$ and such that on termination the configuration is $(\mathrm{ID}_0, (u_{n^k-1}, u_{n^k-2}, \ldots, u_1, u_0))$ (the top of the stack is to the right). The computation of M always encodes the top n^k stack symbols in the simulated computation of ρ_0 on input $\mathcal{A}$ and also the IDs from which the stack symbols are pushed, as is the case above. The ID ID$_0$ is now erased from M's work-tape.

The Turing machine M now simulates the computation of ρ_1 starting from the ID ID_0 and where the top n^k stack elements are taken as $(u_{n^k-1}, u_{n^k-2}, \ldots, u_1, u_0)$. Whenever the top stack element is popped from the stack, this element and the ID from which this symbol is pushed are erased from M's work-tape and a new ID, ID$'$, and a new element, u', are guessed. The computation of M now verifies that either:

- the ID ID$'$ is such that the instruction associated with ID$'$ is a push, and the symbol to be pushed is u';
- there exists a computation of ρ_0 on input $\mathcal{A}$ from the configuration $[\mathrm{ID}', \epsilon]$ to the configuration $[\mathrm{ID}_{n^k-1}, (u')]$,

whence M has the sequence:

$$\mathrm{ID}', u', \mathrm{ID}_{n^k}, u'_{n^k-1}, \mathrm{ID}_{n^k-1}, \ldots, u'_1, \mathrm{ID}_1$$

stored on its work-tape; or

- the ID ID$'$ is the initial ID of ρ_0;
- there exists a computation of ρ_0 on input $\mathcal{A}$ from the configuration $[\mathrm{ID}', \epsilon]$ to the configuration $[\mathrm{ID}_{n^k-1}, \epsilon]$,

whence M has the sequence:

$$\mathrm{ID}', \mathrm{ID}_{n^k}, u'_{n^k-1}, \mathrm{ID}_{n^k-1}, \ldots, u'_1, \mathrm{ID}_1$$

stored on its work-tape. Note that in both of the above cases, we cannot assume that the u'_i's are the same as the u_i's as deep-pushes in the interim computation of ρ_1 might have changed some of the stack elements. In the latter case, when an element is popped from the stack in the computation of ρ_0, the simulation by M no longer guesses a new ID and a new element but just continues with the simulation until either an accept instruction or a reject instruction is reached (ensuring that no deep-pushes are made by ρ_1 to locations below the bottom of the stack). In the former case, the simulation by M simply proceeds as directed above. It is easily seen that M simulates ρ and that the result follows. □

The following corollary is immediate from Corollary 10 and Proposition 12.

Corollary 13 $\mathrm{NPSDS}_s^{+b}(\overline{o}u\overline{d}, o\overline{u}d) = \mathrm{NPSDS}_s(u\overline{o}\overline{d}, o\overline{u}d) = \mathbf{PSPACE}$.

6 Within EXPTIME

Hitherto, we have been focussing on specific restrictions of NPSDS_s. We now examine the computational complexity of problems accepted by arbitrary program schemes of NPSDS_s. Our basic technique in the proof of Proposition 16 is inspired by that used in [1] to show that basic program schemes with an ordinary stack only accept problems in **P**, which in turn was inspired by Cook's construction involving pushdown automata in [3]. We require some definitions before we present our proof.

Definition 14 Let ρ be a program scheme of NPSDS$_s$, where all deep-pushes occur with an index of at most k, and let $\mathcal{A}$ be an input structure of size n. An *extended ID* $(\alpha, \mathbf{s})$ of ρ on input $\mathcal{A}$ is an ID α together with a tuple $\mathbf{s}$ of between 0 and n^k elements of $|\mathcal{A}|$. A pair of extended IDs, $((\alpha, \mathbf{s}), (\beta, \mathbf{t}))$, is called *realizable* if there is a computation of ρ on $\mathcal{A}$ starting in configuration $[\alpha, \mathbf{s}]$ and ending in configuration $[\beta, \mathbf{t}]$, where $|\mathbf{s}| = |\mathbf{t}|$ and the stack associated with any interim configuration has height at least $|\mathbf{s}|$.

Definition 15 Let $((\alpha, \mathbf{s}), (\beta, \mathbf{t}))$ and $((\alpha', \mathbf{s}'), (\beta', \mathbf{t}'))$ be two pairs of extended IDs. These pairs *yield* the pair of extended IDs $((\alpha, \mathbf{s}), (\beta'', \mathbf{t}''))$ if one of the following two rules can be applied.

(a) The instruction associated with β is a push, and the execution of this instruction transforms the configuration $[\beta, \mathbf{t}]$ into the configuration:

- $[\alpha', \mathbf{s}']$, if $|\mathbf{t}| < n^k$;
- $[\alpha', (u_0, \mathbf{s}')]$, if $|\mathbf{t}| = n^k$ (in which case $\mathbf{t}$ is a prefix of $(u_0, \mathbf{s}')$).

The instruction associated with β' is a pop, and the execution of this instruction transforms the configuration $[\beta', \mathbf{t}']$ or $[\beta', (u_0, \mathbf{t}')]$ (depending upon whether $|\mathbf{t}| < n^k$ or $|\mathbf{t}| = n^k$, respectively, above) to the configuration $[\beta'', \mathbf{t}'']$.

(b) $(\beta, \mathbf{t}) = (\alpha', \mathbf{s}')$ and either $(\beta'', \mathbf{t}'') = (\beta', \mathbf{t}')$ or the instruction associated with β' is neither a pop nor a push and execution of this instruction transforms the configuration $[\beta', \mathbf{t}']$ to the configuration $[\beta'', \mathbf{t}'']$.

We refer to rules (a) and (b) as the *yield rules*.

We are now in a position to prove the main result of this section.

Proposition 16 Let ρ be some program scheme of NPSDS$_s$ and let $\mathcal{A}$ be some input structure. Any pair of realizable extended IDs of ρ on input $\mathcal{A}$ can be obtained from the set $Y = \{((\alpha, \mathbf{s}), (\alpha, \mathbf{s})) : (\alpha, \mathbf{s})$ is an extended ID of ρ on input $\mathcal{A}\}$ by iteratively applying the yield rules.

Proof Let $((\alpha, \mathbf{s}), (\beta, \mathbf{t}))$ be a realizable pair of extended IDs where the associated computation χ (of ρ on input $\mathcal{A}$) from configuration $[\alpha, \mathbf{s}]$ to configuration $[\beta, \mathbf{t}]$ has length m. We proceed by induction on m.

Suppose that $m = 1$. As $((\alpha, \mathbf{s}), (\beta, \mathbf{t}))$ is realizable, the instruction associated with α is neither a pop nor a push. Thus, $((\alpha, \mathbf{s}), (\alpha, \mathbf{s}))$ and $((\alpha, \mathbf{s}), (\alpha, \mathbf{s}))$ yield $((\alpha, \mathbf{s}), (\beta, \mathbf{t}))$ by applying rule (b).

Suppose as our induction hypothesis that whenever the associated computation of any realizable pair of extended IDs has length less than m, the realizable pair can be obtained from Y by applying the yield rules. Let the penultimate configuration in χ be $[\beta', \mathbf{t}']$. There are two cases: when the instruction associated with β' is neither a pop nor a push; and when the instruction associated with β' is a pop (note that it cannot be a push as $((\alpha, \mathbf{s}), (\beta, \mathbf{t}))$ is a realizable pair).

In the first case, $|\mathbf{t}'| = |\mathbf{s}| = |\mathbf{t}|$ and $((\alpha, \mathbf{s}), (\beta', \mathbf{t}'))$ is a realizable pair (witnessed by the sub-computation of χ). By the induction hypothesis, $((\alpha, \mathbf{s}), (\beta', \mathbf{t}'))$ can be

obtained from Y by applying the yield rules. An additional application of rule (b) yields that $((\alpha, \mathbf{s}), (\beta, \mathbf{t}))$ can be obtained from Y by applying the yield rules.

Suppose that the instruction associated with β' is a pop. If $|\mathbf{t}'| = n^k + 1$ then define $\mathbf{t}'_0$ such that $\mathbf{t}' = (u_0, \mathbf{t}'_0)$, otherwise define $\mathbf{t}'_0 = \mathbf{t}'$ (recall, stacks are written so that the head of the stack is to the right of the sequence; thus, u_0 is at the bottom of the stack). Let $[\gamma, \mathbf{w}]$ be the (unique) configuration of χ where: $|\mathbf{w}| = |\mathbf{t}'|$; the height of the stack associated with any configuration of χ coming after $[\gamma, \mathbf{w}]$, apart from the last, is at least $|\mathbf{w}|$; and the height of the stack associated with the configuration immediately before $[\gamma, \mathbf{w}]$, namely the configuration $[\gamma', \mathbf{w}']$, is $|\mathbf{w}| - 1$ (that is, $[\gamma, \mathbf{w}]$ is the configuration obtained after executing the push corresponding to the final pop). If $|\mathbf{w}| = n^k + 1$ then define $\mathbf{w}_0$ such that $\mathbf{w} = (u_0, \mathbf{w}_0)$, otherwise define $\mathbf{w}_0 = \mathbf{w}$. In particular, $((\gamma, \mathbf{w}_0), (\beta', \mathbf{t}'_0))$ is a realizable pair, as is $((\alpha, \mathbf{s}), (\gamma', \mathbf{w}'))$, with the induction hypothesis applying to these pairs. An additional application of rule (a) now yields that $((\alpha, \mathbf{s}), (\beta, \mathbf{t}))$ can be obtained from Y by applying the yield rules. The result follows by induction. □

An instance of the *path system problem* consists of: a set of places P; an initial subset of places $I \subseteq P$; a subset of terminal places $T \subseteq P$; and a ternary relation R. The relation R encodes a set of *rules* via: if $u, v \in P$ have already been yielded and $R(u, v, w)$ holds then w becomes yielded. An instance is a yes-instance if we can show that starting from the set of initial places I as our set of yielded places, iteratively applying the rules to yield more places results in a terminal place of T eventually being yielded. The path system problem has long been known to be complete for **P**, and can be solved by a trivial algorithm which simply iteratively computes more and more yielded places.

The complexity class **EXPTIME** consists of those problems solvable by a deterministic algorithm running in time $O(2^{p(n)})$, for some polynomial $p(n)$.

Corollary 17 Any problem accepted by a program scheme of NPSDS$_s$ is in **EXPTIME**.

Proof Let ρ be some program scheme of NPSDS$_s$. By Proposition 16, the problem of deciding whether some input structure $\mathcal{A}$, of size n, is accepted by ρ can be reduced to an instance of the path system problem with an initial set of places $\{((\alpha, \mathbf{s}), (\alpha, \mathbf{s})) : (\alpha, \mathbf{s})$ is an extended ID of ρ on input $\mathcal{A}\}$. However, this instance has size $O(n^{n^m})$, for some constant m. Nevertheless, it is not difficult to see that we can reduce the problem accepted by the program scheme ρ to the path system problem and then solve the path system problem so as to obtain an exponential-time algorithm. □

7 Conclusions

In this paper, we have initiated the study of a high-level model of computation in which there is access to a deep pushdown stack, and we have managed to capture the complexity classes **NP** and **PSPACE** by restricting our model whilst showing that an arbitrary program scheme accepts a problem in **EXPTIME**. We close with some directions for further research.

Whilst we have shown that there is a considerable increase in computational power obtained by replacing stacks with deep pushdown stacks in a basic class of program

schemes (assuming $\mathbf{P} \neq \mathbf{PSPACE}$!), we have as yet been unable to ascertain exactly the computational complexity of the problems accepted by program schemes of NPSDS_s. At the moment, all we know is that $\mathbf{PSPACE} \subseteq \mathrm{NPSDS}_s \subseteq \mathbf{EXPTIME}$.

Looking at Example 8 where we show that the complement of the Hamiltonian path problem is accepted by a program scheme of NPSDS_s, we can see no obvious restriction of the general class of program schemes so that we capture the complexity class **co-NP** (the complement of **NP**). Note that our program scheme in Example 8 is clearly in $\mathrm{NPSDS}_s(\overline{o}u\overline{d}, o\overline{u}d)$ yet by Corollary 13, $\mathrm{NPSDS}_s(\overline{o}u\overline{d}, o\overline{u}d)$ captures **PSPACE**. It would be interesting to capture **co-NP**, and more generally the classes of the Polynomial Hierarchy, by restricting the program schemes of NPSDS_s.

Let us comment on the height of the stack in different circumstances. As can be seen from the proof of Proposition 6, any problem in **NP** can be accepted by a program scheme of $\mathrm{NPSDS}_s^{+b}(ou\overline{d}, \overline{o}\overline{u}d, ou\overline{d})$ where the maximum height of the stack is only ever polynomial in the size of the input structure. From Example 8, the problem co-HP can be accepted by a program scheme of $\mathrm{NPSDS}_s(\overline{o}u\overline{d}, o\overline{u}d)$ so that the height of the stack is always $O(n^{n+1})$. From the proof of Proposition 9, the problem accepted by a Turing machine running in space $O(n^k)$ and time $O(f(n))$ can be accepted by a program scheme of $\mathrm{NPSDS}_s(\overline{o}u\overline{d}, o\overline{u}d)$ so that the height of the stack is always $O(n^k \cdot f(n)) = O(n^k \cdot c^{n^k})$, for some c. It would be interesting to try and relate the height of the stack used in a computation with the complexity of the problem in hand.

Finally, one might vary the mechanism by which locations within the stack are accessed. For example, one might have two sorts, a 'numeric sort' and an 'element sort', with values from the numeric sort being manipulable and used to access locations within the stack. This set-up might allow access to locations 'exponentially deep' within the stack (should the numeric values be such that exponential numbers can be created). Also, one might allow deep-pops within the current model; thus, the top 'polynomial' portion of the stack can be used as an array. Finally, one might consider a model where elements (or even sequences of elements) can be inserted within the stack at some location.

References

[1] A.A. Arratia-Quesada, S.R. Chauhan and I.A. Stewart, Hierarchies in classes of program schemes, *Journal of Logic and Computation* **9** (1999) 915–957.

[2] R. Constable and D. Gries, On classes of program schemata, *SIAM Journal of Computing* **1** (1972) 66–118.

[3] S.A. Cook, Characterizations of pushdown pmachines in terms of time-bounded computers, *Journal of the Association for Computing Machinery* **18** (1971) 4–18.

[4] E. Dahlhaus, Reduction to NP-complete problems by interpretations, *Proc. of Logic and Machines: Decision Problems and Complexity* (E. Börger, G. Hasenjaeger, D. Rödding, eds.), Lecture Notes in Computer Science Vol. 171, Springer (1984) 357–365.

[5] H.-D. Ebbinghaus and J. Flum, *Finite Model Theory*, Springer-Verlag (1995).

[6] H. Friedman, Algorithmic procedures, generalized Turing algorithms and elementary recursion theory, *Logic Colloquium* 1969 (R.O. Gandy, C.M.E. Yates, eds.), North-Holland (1971) 361–390.

[7] S. Ginsberg and E. Spanier, Finite-turn pushdown automata, *SIAM Journal on Control* **4** (1968) 429–453.

[8] D. Harel and D. Peleg, On static logics, dynamic logics, and complexity classes, *Information and Control* **60** (1984) 86–102.

[9] N. Immerman, *Descriptive Complexity*, Springer (1999).

[10] N.D. Jones and S.S. Muchnik, Even simple programs are hard to analyze, *Journal of Association for Computing Machinery* **24** (1977) 338–350.

[11] T. Kasai, An hierarchy between context-free and context-sensitive languages, *Journal of Computer and System Sciences* **4** (1970) 492-508.

[12] L. Libkin, *Elements of Finite Model Theory*, Springer (2004).

[13] A. Meduna, *Automata and Languages*: *Theory and Applications*, Springer (2000).

[14] A. Meduna, Simultaneously one-turn two-pushdown automata, *International Journal of Computer Mathematics* **80** (2003) 679–687.

[15] A. Meduna, Deep pushdown automata, *Acta Informatica* **42** (2006) 541–552.

[16] M. Paterson and N. Hewitt, Comparative schematology, *Record of Project MAC Conf. on Concurrent Systems and Parallel Comput.*, ACM Press (1970) 119–128.

[17] L. Spector, J. Klein and M. Keijzer, The Push3 execution stack and the evolution of control, *Proc. of Genetic and Evolutionary Computation Conference* (H.-G. Beyer, U.-M. O'Reilly, eds.), ACM Press (2005) 1689–1696.

[18] I.A. Stewart, Using the Hamiltonian path operator to capture NP, *Journal of Computer and System Sciences* **45** (1992) 127–151.

[19] I.A. Stewart, Program schemes, arrays, Lindström quantifiers and zero-one laws, *Theoretical Computer Science* **275** (2002) 283–310.

[20] I.A. Stewart, Using program schemes to logically capture polynomial-time on certain classes of structures, *London Mathematical Society Journal of Computation and Mathematics* **6** (2003) 40–67.

[21] I.A. Stewart, Logical and complexity-theoretic aspects of models of computation with restricted access to arrays, *Proc. of Computation and Logic in the Real World, Third Conference on Computability in Europe* (*CiE* 2007) (ed. S.B. Cooper, T.F. Kent, B. Löwe, A. Sorbi) (2007) 324–331.

[22] J. Tiuryn and P. Urzyczyn, Some relationships between logics of programs and complexity theory, *Theoretical Computer Science* **60** (1988) 83–108.

[23] L.G. Valiant, The equivalence problem for deterministic finite turn pushdown automata, *Information and Control* **81** (1989) 265–279.